PILOT-02: Orchestrator Skeleton (Go)

🎯 CONTEXT

Phase: PILOT (Week 0)
Component: Orchestrator - Basic HTTP Server
Estimated Time: 30 min AI execution + 20 min verification
Complexity: High
Risk Level: HIGH (critical component, tests Go generation)

📦 DEPENDENCIES

Must Complete First:

- **PILOT-01:** Bootstrap project structure ✅ COMPLETED

Required Services Running:


bash

```
# Verify databases are running
make verify
# Expected: All 4 services HEALTHY
```

Required Environment:


bash

```
# Go installed
go version  # Go 1.21+

# Project structure exists
ls services/orchestrator/
```

🎯 OBJECTIVE

Create a **Go-based HTTP server** that will serve as the orchestrator for all agents. This is the foundation for agent registration, request routing, and coordination.

Success Criteria:

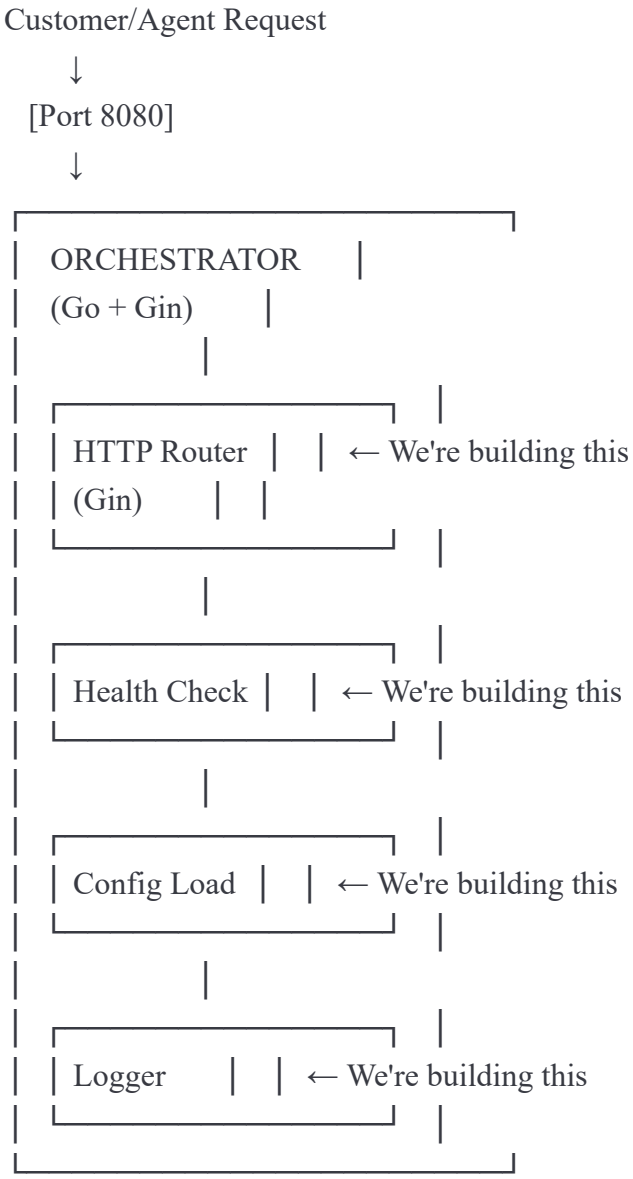
- ✅ Go application compiles without errors
- ✅ HTTP server starts on port 8080
- ✅ `/health` endpoint returns 200 OK
- ✅ Structured logging works (JSON format)
- ✅ Configuration loading works (from environment)
- ✅ Docker image builds successfully (< 50 MB)
- ✅ Basic tests pass

Failure Signs:

- ❌ Go code doesn't compile
- ❌ Server crashes on startup
- ❌ Health endpoint returns errors
- ❌ Docker build fails
- ❌ Memory leaks or high CPU usage

📐 TECHNICAL SPECIFICATION

Architecture Context



File Structure to Create

```
services/orchestrator/
├── cmd/
│   └── orchestrator/
│       └── main.go            # CREATE: Entry point
│
├── internal/
│   ├── config/
│   │   └── config.go          # CREATE: Configuration
│   ├── handlers/
│   │   └── health.go          # CREATE: Health check handler
│   └── logger/
│       └── logger.go          # CREATE: Structured logging
│
├── pkg/
│   └── version/
│       └── version.go         # CREATE: Version info
│
├── go.mod                     # CREATE: Go module
├── go.sum                     # CREATE: Dependencies (auto-generated)
├── Dockerfile                 # CREATE: Multi-stage build
├── .dockerignore              # CREATE: Docker ignore
├── Makefile                   # CREATE: Build commands
└── README.md                  # CREATE: Documentation
```

💻 IMPLEMENTATION REQUIREMENTS

1. go.mod (Module Definition)

go

```
module github.com/yourorg/optiinfra/orchestrator

go 1.21

require (
    github.com/gin-gonic/gin v1.10.0
    github.com/joho/godotenv v1.5.1
    go.uber.org/zap v1.26.0
)
```

2. cmd/orchestrator/main.go (Entry Point)

go

```go
package main

import (
    "context"
    "fmt"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/yourorg/optiinfra/orchestrator/internal/config"
    "github.com/yourorg/optiinfra/orchestrator/internal/handlers"
    "github.com/yourorg/optiinfra/orchestrator/internal/logger"
)

func main() {
    // Initialize logger
    log := logger.NewLogger()
    defer log.Sync()

    log.Info("Starting OptiInfra Orchestrator")

    // Load configuration
    cfg, err := config.Load()
    if err != nil {
        log.Fatal("Failed to load configuration", "error", err)
    }

    // Set Gin mode
    if cfg.Environment == "production" {
        gin.SetMode(gin.ReleaseMode)
    }

    // Create router
    router := gin.New()
    router.Use(gin.Recovery())

    // Use custom logger middleware
    router.Use(func(c *gin.Context) {
```

```
    start := time.Now()
    path := c.Request.URL.Path
    c.Next()

    log.Info("HTTP request",
        "method", c.Request.Method,
        "path", path,
        "status", c.Writer.Status(),
        "duration_ms", time.Since(start).Milliseconds(),
    )
})

// Register routes
router.GET("/health", handlers.HealthCheck)
router.GET("/", func(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{
        "service": "OptiInfra Orchestrator",
        "version": "0.1.0",
        "status": "running",
    })
})

// Create server
srv := &http.Server{
    Addr:    fmt.Sprintf(":%d", cfg.Port),
    Handler: router,
}

// Start server in goroutine
go func() {
    log.Info("Server starting", "port", cfg.Port)
    if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
        log.Fatal("Server failed to start", "error", err)
    }
}()

// Wait for interrupt signal
quit := make(chan os.Signal, 1)
signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
<-quit
```

```
    log.Info("Shutting down server...")

    // Graceful shutdown
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    if err := srv.Shutdown(ctx); err != nil {
        log.Error("Server forced to shutdown", "error", err)
    }

    log.Info("Server stopped")
}
```

3. internal/config/config.go (Configuration)

go

```go
package config

import (
    "os"
    "strconv"

    "github.com/joho/godotenv"
)

type Config struct {
    Port        int
    Environment string
    LogLevel    string
}

func Load() (*Config, error) {
    // Load .env file if exists (ignore error in production)
    godotenv.Load()

    port := 8080
    if portStr := os.Getenv("ORCHESTRATOR_PORT"); portStr != "" {
        if p, err := strconv.Atoi(portStr); err == nil {
            port = p
        }
    }

    return &Config{
        Port:        port,
        Environment: getEnv("ENVIRONMENT", "development"),
        LogLevel:    getEnv("LOG_LEVEL", "info"),
    }, nil
}

func getEnv(key, defaultValue string) string {
    if value := os.Getenv(key); value != "" {
        return value
    }
    return defaultValue
}
```

4. internal/handlers/health.go (Health Check)

go

```go
package handlers

import (
    "net/http"
    "time"

    "github.com/gin-gonic/gin"
)

type HealthResponse struct {
    Status    string    `json:"status"`
    Timestamp time.Time `json:"timestamp"`
    Version   string    `json:"version"`
    Uptime    string    `json:"uptime"`
}

var startTime = time.Now()

func HealthCheck(c *gin.Context) {
    uptime := time.Since(startTime)

    response := HealthResponse{
        Status:    "healthy",
        Timestamp: time.Now(),
        Version:   "0.1.0",
        Uptime:    uptime.String(),
    }

    c.JSON(http.StatusOK, response)
}
```

5. internal/logger/logger.go (Structured Logging)

go

```go
package logger

import (
    "os"

    "go.uber.org/zap"
    "go.uber.org/zap/zapcore"
)

type Logger struct {
    *zap.SugaredLogger
}

func NewLogger() *Logger {
    // Determine log level
    logLevel := zapcore.InfoLevel
    if level := os.Getenv("LOG_LEVEL"); level != "" {
        switch level {
        case "debug":
            logLevel = zapcore.DebugLevel
        case "warn":
            logLevel = zapcore.WarnLevel
        case "error":
            logLevel = zapcore.ErrorLevel
        }
    }

    // Configure encoder
    encoderConfig := zapcore.EncoderConfig{
        TimeKey:       "timestamp",
        LevelKey:      "level",
        NameKey:       "logger",
        CallerKey:     "caller",
        MessageKey:    "message",
        StacktraceKey: "stacktrace",
        LineEnding:    zapcore.DefaultLineEnding,
        EncodeLevel:   zapcore.LowercaseLevelEncoder,
        EncodeTime:    zapcore.ISO8601TimeEncoder,
        EncodeDuration: zapcore.SecondsDurationEncoder,
        EncodeCaller:  zapcore.ShortCallerEncoder,
    }
```

```
    // Create core
    core := zapcore.NewCore(
        zapcore.NewJSONEncoder(encoderConfig),
        zapcore.AddSync(os.Stdout),
        logLevel,
    )

    // Create logger
    logger := zap.New(core, zap.AddCaller(), zap.AddStacktrace(zapcore.ErrorLevel))

    return &Logger{SugaredLogger: logger.Sugar()}
}
```

6. pkg/version/version.go (Version Info)

go

```
package version

const (
    Version = "0.1.0"
    Service = "OptiInfra Orchestrator"
)
```

7. Dockerfile (Multi-Stage Build)

dockerfile

```dockerfile
# Build stage
FROM golang:1.21-alpine AS builder

WORKDIR /build

# Install dependencies
RUN apk add --no-cache git

# Copy go mod files
COPY go.mod go.sum ./
RUN go mod download

# Copy source code
COPY . .

# Build binary
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build \
    -ldflags="-w -s" \
    -o orchestrator \
    ./cmd/orchestrator

# Runtime stage
FROM alpine:latest

RUN apk --no-cache add ca-certificates

WORKDIR /app

# Copy binary from builder
COPY --from=builder /build/orchestrator .

# Expose port
EXPOSE 8080

# Health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
    CMD wget --no-verbose --tries=1 --spider http://localhost:8080/health || exit 1
```

```
# Run
CMD ["./orchestrator"]
```

8. .dockerignore

```
# Git
.git
.gitignore

# Build artifacts
bin/
*.exe
*.exe~
*.dll
*.so
*.dylib

# Test files
*.test
*.out

# Go workspace
go.work

# IDE
.vscode/
.idea/
*.swp
*.swo

# Docs
README.md
docs/

# Misc
.DS_Store
*.log
```

9. Makefile

makefile

```
.PHONY: help build run test clean docker-build docker-run

# Variables
APP_NAME=orchestrator
DOCKER_IMAGE=optiinfra-orchestrator
VERSION=0.1.0

help:
  @echo "OptiInfra Orchestrator - Build Commands"
  @echo "========================================"
  @echo "make build        - Build Go binary"
  @echo "make run          - Run locally"
  @echo "make test         - Run tests"
  @echo "make clean        - Clean build artifacts"
  @echo "make docker-build - Build Docker image"
  @echo "make docker-run   - Run in Docker"

build:
  @echo "Building $(APP_NAME)..."
  go build -o bin/$(APP_NAME) ./cmd/orchestrator

run:
  @echo "Running $(APP_NAME)..."
  go run ./cmd/orchestrator

test:
  @echo "Running tests..."
  go test -v -race -coverprofile=coverage.out ./...
  go tool cover -html=coverage.out -o coverage.html

clean:
  @echo "Cleaning..."
  rm -rf bin/
  rm -f coverage.out coverage.html

docker-build:
  @echo "Building Docker image..."
  docker build -t $(DOCKER_IMAGE):$(VERSION) -t $(DOCKER_IMAGE):latest .

docker-run:
  @echo "Running Docker container..."
```

```
	docker run -p 8080:8080 --name $(APP_NAME) $(DOCKER_IMAGE):latest

fmt:
	@echo "Formatting code..."
	go fmt ./...

lint:
	@echo "Linting code..."
	go vet ./...

deps:
	@echo "Downloading dependencies..."
	go mod download
	go mod tidy
```

10. cmd/orchestrator/main_test.go (Basic Test)

go

```go
package main

import (
    "net/http"
    "net/http/httptest"
    "testing"

    "github.com/gin-gonic/gin"
    "github.com/yourorg/optiinfra/orchestrator/internal/handlers"
)

func TestHealthCheck(t *testing.T) {
    // Setup
    gin.SetMode(gin.TestMode)
    router := gin.New()
    router.GET("/health", handlers.HealthCheck)

    // Test
    w := httptest.NewRecorder()
    req, _ := http.NewRequest("GET", "/health", nil)
    router.ServeHTTP(w, req)

    // Assert
    if w.Code != http.StatusOK {
        t.Errorf("Expected status 200, got %d", w.Code)
    }

    // Check JSON response contains "healthy"
    body := w.Body.String()
    if !contains(body, "healthy") {
        t.Errorf("Expected 'healthy' in response, got: %s", body)
    }
}

func contains(s, substr string) bool {
    return len(s) >= len(substr) &&
        (s == substr || len(s) > len(substr) &&
        (s[:len(substr)] == substr || contains(s[1:], substr)))
}
```

11. README.md

markdown

OptiInfra Orchestrator

Go-based orchestrator for coordinating multiple AI agents in the OptiInfra platform.

Features

- HTTP server (Gin framework)
- Structured JSON logging (Zap)
- Health check endpoint
- Configuration from environment
- Graceful shutdown
- Docker support

Development

Prerequisites

- Go 1.21+
- Docker (optional)

Running Locally

```bash
# Install dependencies
go mod download

# Run
make run

# Or
go run ./cmd/orchestrator
```

Building

```bash
# Build binary
make build

# Build Docker image
make docker-build
```

Testing
```bash
# Run tests
make test

# View coverage
open coverage.html
```

API Endpoints

GET /health

Health check endpoint.

Response:
```json
{
  "status": "healthy",
  "timestamp": "2025-10-16T10:00:00Z",
  "version": "0.1.0",
  "uptime": "5m30s"
}
```

GET /

Service info endpoint.

Response:
```json
{
  "service": "OptiInfra Orchestrator",
  "version": "0.1.0",
  "status": "running"
}
```

Configuration

Environment variables:

- `ORCHESTRATOR_PORT` - Port to listen on (default: 8080)
- `ENVIRONMENT` - Environment name (default: development)
- `LOG_LEVEL` - Log level: debug, info, warn, error (default: info)

Docker
```bash
# Build image
docker build -t optiinfra-orchestrator .

# Run container
docker run -p 8080:8080 optiinfra-orchestrator

# Or use docker-compose (from project root)
docker-compose up orchestrator
```

Architecture

main.go ↓ config.Load() # Load environment variables ↓ logger.New() # Initialize structured logger ↓ gin.New() # Create HTTP router ↓ RegisterRoutes() # Register endpoints ↓ srv.Start() # Start HTTP server ↓ GracefulShutdown() # Wait for SIGTERM

Next Steps

After this pilot phase:
- Add agent registry (0.6)
- Add request routing (0.7)
- Add coordination logic (0.8)
- Add authentication
- Add metrics (Prometheus)

✅ VALIDATION COMMANDS

Step 1: Verify Go Environment

bash

```
cd services/orchestrator

# Check Go version
go version
# Expected: go version go1.21.x or higher

# Check module
cat go.mod
# Expected: module definition with correct dependencies
```

Step 2: Download Dependencies

bash

```
# Download all dependencies
go mod download

# Verify dependencies
go mod verify
# Expected: all modules verified

# Tidy up
go mod tidy
```

Step 3: Build Application

bash

```bash
# Build binary
go build -o bin/orchestrator ./cmd/orchestrator

# Check binary exists
ls -lh bin/orchestrator
# Expected: executable file ~10-20 MB

# Or use Makefile
make build
```

Step 4: Run Tests

```bash
# Run all tests
go test -v ./...

# Expected output:
# === RUN   TestHealthCheck
# --- PASS: TestHealthCheck (0.00s)
# PASS
# ok      github.com/yourorg/optiinfra/orchestrator/cmd/orchestrator

# Check coverage
go test -cover ./...
# Expected: >70% coverage
```

Step 5: Start Server (Local)

```bash
```

```
# Start server
./bin/orchestrator

# Or
make run

# Expected output:
# {"level":"info","timestamp":"2025-10-16T10:00:00Z","message":"Starting OptiInfra Orchestrator"}
# {"level":"info","timestamp":"2025-10-16T10:00:00Z","message":"Server starting","port":8080}
```

Step 6: Test Health Endpoint

In another terminal:


bash

```bash
# Test health endpoint
curl http://localhost:8080/health

# Expected output:
# {
#   "status": "healthy",
#   "timestamp": "2025-10-16T10:00:00Z",
#   "version": "0.1.0",
#   "uptime": "5s"
# }

# Test root endpoint
curl http://localhost:8080/

# Expected output:
# {
#   "service": "OptiInfra Orchestrator",
#   "version": "0.1.0",
#   "status": "running"
# }

# Check response code
curl -I http://localhost:8080/health
# Expected: HTTP/1.1 200 OK
```

Step 7: Build Docker Image


bash

```bash
# Stop local server first (Ctrl+C)

# Build Docker image
docker build -t optiinfra-orchestrator:latest .

# Expected output:
# Successfully built [image-id]
# Successfully tagged optiinfra-orchestrator:latest

# Check image size
docker images optiinfra-orchestrator
# Expected: < 50 MB (Alpine-based)

# Or use Makefile
make docker-build
```

Step 8: Run in Docker

bash

```bash
# Run container
docker run -d -p 8080:8080 --name orchestrator optiinfra-orchestrator:latest

# Check container is running
docker ps | grep orchestrator
# Expected: Container running

# Check logs
docker logs orchestrator
# Expected: Startup logs, no errors

# Test health endpoint
curl http://localhost:8080/health
# Expected: {"status":"healthy",...}

# Stop container
docker stop orchestrator
docker rm orchestrator
```

Step 9: Integration with docker-compose

bash

```
# Go back to project root
cd ../../

# Update docker-compose.yml to uncomment orchestrator service
# Start with docker-compose
docker-compose up orchestrator

# In another terminal, test
curl http://localhost:8080/health
# Expected: {"status":"healthy",...}

# Stop
docker-compose down
```

Step 10: Final Verification

bash

```
# Format code
cd services/orchestrator
go fmt ./...

# Lint code
go vet ./...
# Expected: No issues

# Final test
make test
# Expected: All tests pass
```

🎯 SUCCESS CRITERIA CHECKLIST

After running all validation commands, verify:

- [ ] Go code compiles without errors
- [ ] `go build` produces binary (10-20 MB)
- [ ] All tests pass (`go test -v ./...`)
- [ ] Test coverage > 70%
- [ ] Server starts locally without errors
- [ ] `/health` endpoint returns 200 OK
- [ ] `/health` response contains "healthy"
- [ ] `/` endpoint returns service info
- [ ] Structured logging outputs JSON
- [ ] Configuration loads from environment
- [ ] Docker image builds successfully
- [ ] Docker image size < 50 MB
- [ ] Container runs without errors
- [ ] Container health check passes
- [ ] Graceful shutdown works (Ctrl+C)
- [ ] No Go vet warnings
- [ ] No obvious security issues

Expected Time: < 50 minutes total (30 min generation + 20 min verification)

🚨 TROUBLESHOOTING

Issue 1: Go build fails with "package not found"

Solution:

bash

```
# Re-download dependencies
go mod download
go mod tidy

# Verify go.mod paths match your actual paths
cat go.mod | grep module
# Should match your import statements
```

Issue 2: Server won't start - "port already in use"

Solution:

bash

```
# Check what's using port 8080
lsof -i :8080

# Kill the process or change port
export ORCHESTRATOR_PORT=8081
go run ./cmd/orchestrator
```

Issue 3: Docker build fails

Solution:


bash

```
# Check Docker daemon
docker info

# Try with --no-cache
docker build --no-cache -t optiinfra-orchestrator .

# Check Dockerfile syntax
docker build --progress=plain -t optiinfra-orchestrator .
```

Issue 4: Health check returns 404

Solution:


bash

```
# Verify server is running
ps aux | grep orchestrator

# Check which port it's listening on
netstat -an | grep LISTEN | grep 8080

# Try with explicit localhost
curl http://127.0.0.1:8080/health
```

Issue 5: Tests fail

Solution:

bash

```
# Run with verbose output
go test -v ./...

# Run specific test
go test -v -run TestHealthCheck ./cmd/orchestrator

# Check for race conditions
go test -race ./...
```

Issue 6: Import cycle detected

Solution:

bash

```
# This means your imports have circular dependencies
# Restructure packages to remove cycles

# Check import graph
go mod graph
```

📦 DELIVERABLES

This prompt should generate:

1. **Go Source Files (6 files):**
   - cmd/orchestrator/main.go
   - internal/config/config.go
   - internal/handlers/health.go
   - internal/logger/logger.go
   - pkg/version/version.go
   - cmd/orchestrator/main_test.go
2. **Configuration Files:**
   - go.mod
   - go.sum (auto-generated)
3. **Docker Files:**

    - Dockerfile
    - .dockerignore
4. **Build Files:**
    - Makefile
5. **Documentation:**
    - README.md
6. **Working HTTP Server:**
    - Compiles and runs
    - Health check endpoint
    - Structured logging
    - Docker support

🎯 NEXT STEPS

After this prompt succeeds:

1. ✅ **Verify:** Server running, health check works
2. ✅ **Commit:** `git add . && git commit -m "PILOT-02: Orchestrator skeleton"`
3. ➡️ **Continue:** PILOT-03 (Cost Agent Skeleton - FastAPI)

What we'll add later (Foundation phase):

- Agent registry (Prompt 0.6)
- Request routing (Prompt 0.7)
- Coordination logic (Prompt 0.8)
- Redis integration
- PostgreSQL integration

📝 NOTES FOR WINDSURF

IMPORTANT INSTRUCTIONS:

1. **Use exact import paths** - Match go.mod module name
2. **Generate complete files** - No "TODO" or placeholders
3. **Follow Go conventions** - Proper naming, formatting
4. **Multi-stage Docker build** - Keep image small (< 50 MB)
5. **Structured logging** - JSON format with Zap
6. **Graceful shutdown** - Handle SIGTERM properly
7. **Health checks** - Return useful information
8. **Tests included** - Basic test coverage

DO NOT:

- Use deprecated packages
- Skip error handling
- Use global variables unnecessarily
- Forget to add go.mod dependencies
- Make image larger than needed
- Skip graceful shutdown

EXECUTE ALL TASKS. CREATE COMPLETE, WORKING GO APPLICATION. THIS PROVES WINDSURF CAN HANDLE GO CODE GENERATION.